**COMMENTS ABOUT THIS RELEASE OF DOCUMENT:**

**CONDITIONAL JUMPS**

In class, I was asked about the conditional jump instructions. The student asked if these tested the condition of the last triple. I said that they explicitly referenced a triple for the test but then was asked as to why this was necessary. I stumbled on it and then said that either way was possible and I would tell you which I had implemented. The answer is that we actually do reference a triple for the tested result. A sample of the form is **JZ -10 -6** This would jump to triple 6 if the result of the operation in triple 10 is zero. Why then must we reference an explicit triple since in all our samples the triple tested is always immediately preceding the test? The reason, though not evident in this assignment, is optimization. We may find that a calculation is not needed because precisely the same computation has already occurred in a prior triple and that triple’s result is not been affected by any intervening triples. A good optimization strategy would be to avoid recalculating the result and just referencing the previously computed triple. Such a triple could be an arbitrary distance from the test, not necessarily in the previous triple.

**THE VARIABLE forwardTest**

This variable is set before a test (the computation and conditional jump) to indicate whether the jump is backwards or forwards. Forward jumps jump ahead if the test condition is NOT met. Backward jumps are done when the condition is met. In my desire to show you the use of embedded actions (embedded in the rhs of grammatical rules) versus the sue of non-terminals that always derive the string of length zero, I got sloppy and places such a setting in the wrong place. Please use the grammar that was placed out there Saturday night at about 11PM, and none earlier than that.

**THE COMPILER TASK AND ITS LANGUAGE**

You are to write a recursive descent parser for a simple language that has just four statement types (**assignment**, **while**, **do..while** and **if**). The language also allows groups of statements to be treated as a block by enclosing then in braces, much like C does.

The precise syntax is defined by a grammar that I have written in Bison. These are the essential rules for a well-formed program.

All programs start with a list of variables, separated by commas and terminated by a semicolon. Variables are global and they are all of type int, so no actual typing occurs. There is no other way to declare variables. Variables must be alphanumeric and start with an alphabetic character. The variable name is optionally followed by a vector dimension (the vector’s bound), which must be a single positive integer. This declares the number of component for the vector. If there is not subscript bound, then the variable denotes a scalar. Case does matter; so variable B differs from variable b. Although a variable name can be as long as you want, only the first eight letters distinguish it. Thus, the names **cop3402a** and **cop3402assignments** are treated as the same variable name since the start with the same first eight characters.

Typical declarations are

**a, b, c[7];**

**vector3d[3], upper, lower;**

The body of the program follows the declaration of variables. The body consists of a sequence of statements terminated by an end, that is, a statement that says “end.” Again this is a case sensitive language so end is not the same as END.

The simplest program is

**;**

**end**

This has no declared variables. Notice that the semicolon is still required to terminate the null list of declarations. It has no body as well.

The statement syntax and semantics are as follows.

An **assignment** statement consists of a variable, with subscript if this denotes a vector, followed by an assignment operator (:=) and then an arithmetic expression and a terminating semicolon. Expressions involve the binary operators +, -, \*, / and parentheses to override normal precedence and associativity. The semantics is normal in that the rhs is evaluated and its result is assigned to the scalar or vector component on the left.

The **while** statement has the syntax **while test do statement** where **statement** can be a simple statement or a block (sequence of statement surrounded by an open and close brace). The test is a pair of arithmetic expressions separated by a relational operator. The relational operators allowed are **=**, **#**, **>**, **<**, **>=** and **<=**. The semantics is as you would expect.

The **do** statement has the syntax **do { statement.list } while test** where **statement.list** is a sequence of statements. The test is a pair of arithmetic expressions separated by a relational operator. The relational operators allowed are **=**, **#**, **>**, **<**, **>=** and **<=**. The semantics is as you would expect.

The **if** statement is of the form **if test then statement1 [else statement2].** The **test** has the same syntax as with a **while**.

Your goal is to compile programs in this simple language, generating intermediate code in the form of triples. The solution I presented has its lexical analyzer done in FLEX and its parser in BISON. This is a reference solution, but it is complete, if not always obvious. I have used some C-code that is a bit cryptic but that you will see in real life. I will discuss that.

The code generated is dumped after each line in an incremental fashion. That is, no code is echoed more than once in these embedded reports. The full code is reported again at the end of the program parse. Additionally, the final report must state the references to each variable, differentiating read from write references.

The code generated is based on a simple machine model where each instruction is a triple consisting of an operator and two operands. Any operand that is not required is set to 0. All instructions are numbered starting at 1. All variables are numbered starting at 1. References to symbol table entries are always positive (positions start at 1). References to instruction locations always have a negative number in front of them in the generated code.

The opcodes can be output in upper case, lower case or mixed case. These are:

 **Constant term**

CON op1 0 // op1 is a constant value (positive or zero)

 **Assignment operators**

 := op1 op2 // assign value from op2 to scalar op1 (op2 is a –triple)

 []= op1 op2 // assigns the value from op3 to vector component op1[op2]

 \_ op3 0

 **Accessor for subscripted variable**

 =[] op1 op2 // evaluates to op1[op2] (used in rhs references)

 **Arithmetic operations**

+ op1 op2 // operands can be to symbol table (positive)

- op1 op2 // operands can be references to other triples (negative)

\* op1 op2 // no operand can be zero

/ op1 op2

**Unconditional jump uses first field as destination triple**

JUMP -loc 0 // jumps to triple at loc

**Conditional jumps use first field to reference triple where test value was computed**

**Second field is destination triple**

JZ -loc1 -loc2 // conditional jump on zero (=0)

JNZ -loc1 -loc2 // conditional jump on not zero (#0)

JP -loc1 -loc2 // conditional jump on positive (>0)

JNP -loc1 -loc2 // conditional jump on not positive (<=0)

JM -loc1 -loc2 // conditional jump on minus (<0)

JNM -loc1 -loc2 // conditional jump on not minus (>=0)

**Miscellaneous**

ERROR 0 0 // error (can extend to put error codes in triple)

EXIT 0 0 // all done

%token SEMICOLON COMMA END LBRACE RBRACE LBRACK RBRACK

%token RPAREN LPAREN NUMBER IDENT WHILE DO ASSIGN BAD

%token IF THEN ELSE

%nonassoc RELOP

%left PLUS MINUS

%left TIMES DIVIDE

%{

/\*////////////////////////////////////////////////////////////////////////

// YACC for a simple language //

// //

// FILENAME: while.y //

// //

// Author(s): Charles E. Hughes //

// //

////////////////////////////////////////////////////////////////////////\*/

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <math.h>

#include <ctype.h>

#include <limits.h>

/\* Constants \*/

#define IDLENGTH 9 /\* one more (for \0) than max ident length \*/

#define SYMMAX 20 /\* size of symbol table \*/

#define CODEMAX 100 /\* size of code table \*/

/\* Code table \*/

struct {

 char opcode[IDLENGTH]; /\* mnemonic of operator \*/

 int first,second; /\* first and second operand \*/

} code[CODEMAX]; /\* code table \*/

/\* Symbol table \*/

struct {

 char name[9]; /\* identifier name \*/

 int size; /\* zero for scalar; number of elements for vector \*/

 int countRead; /\* count the read references. \*/

 int countWrite; /\* count the write references \*/

} symtab[SYMMAX]; /\* symbol table \*/

/\* Other variables \*/

char idname[IDLENGTH]; /\* Ident just found -- first eight characters are meaningful \*/

int triple; /\* Next available code slot \*/

int showStart; /\* start of code area yet to be displayed \*/

int symsize; /\* next available symbol table slot \*/

int forwardTest; /\* flag indicating cond jump forward or backward \*/

int errorNo; /\* error number counter \*/

int warnNo; /\* warning number counter \*/

/\* Function declarations \*/

void basicinit(void);

void enter(char \*name, int size);

void backpatch(int trip, int val);

void backpatch1(int trip, int val);

void emit(char \*opcode, int first, int second);

void ok(int extra);

void fatal(char const \*msg);

void dumpcode(void);

void countReport(void);

void error(char const \*s);

void warning(char const \*s);

void yyerror(char const \*s);

%}

%%

program:

 startup decl body END

 { emit("EXIT",0,0); dumpcode();

 showStart = 1; dumpcode(); countReport();

 }

 ;

startup:

 /\* empty \*/ { basicinit(); }

 ;

decl:

 var.list SEMICOLON |

 SEMICOLON |

 error { error("declaration syntax"); ok(COMMA); yyerrok; }

 ;

var.list:

 var.list COMMA var.def |

 var.def |

 error { error("declaration syntax"); ok(COMMA); yyerrok; }

 ;

var.def:

 IDENT LBRACK NUMBER RBRACK

 { if ($1 > 0) warning("duplicate name");

 else enter(idname,$3); } |

 IDENT

 { if ($1 > 0) warning("duplicate name");

 else enter(idname,0);

 }

 ;

body:

 statement.list

 ;

statement.list:

 statement.list statement |

 /\* empty \*/

 ;

statement:

 assign SEMICOLON |

 while |

 do |

 if |

 LBRACE statement.list RBRACE |

 SEMICOLON |

 error { error("declaration syntax"); ok(LBRACE); yyerrok; }

while:

 while.start {forwardTest = 1;} test DO statement

 { emit("JUMP",-$1,0);

 backpatch($3,-triple);

 }

 ;

while.start:

 WHILE { $$ = triple; }

 ;

test:

 exp RELOP exp

 { emit("-",$1,$3);

 $$ = triple;

 switch( (char) $2) {

 case '=':

 if (forwardTest) emit("JNZ",-triple+1,0);

 else emit("JZ",-triple+1,0);

 break;

 case '#':

 if (forwardTest) emit("JZ",-triple+1,0);

 else emit("JNZ",-triple+1,0);

 break;

 case '>':

 if (forwardTest) emit("JNP",-triple+1,0);

 else emit("JP",-triple+1,0);

 break;

 case '<':

 if (forwardTest) emit("JNM",-triple+1,0);

 else emit("JM",-triple+1,0);

 break;

 case '+':

 if (forwardTest) emit("JM",-triple+1,0);

 else emit("JNM",-triple+1,0);

 break;

 case '-':

 if (forwardTest) emit("JP",-triple+1,0);

 else emit("JNP",-triple+1,0);

 break;

 }

 }

 ;

do:

 do.start LBRACE statement.list RBRACE WHILE {forwardTest=0;} test SEMICOLON

 { backpatch($7, -$1); }

 ;

do.start:

 DO { $$ = triple; }

 ;

if:

 IF {forwardTest=1;} test THEN statement optionalElse

 { backpatch($3, -$6); }

 ;

optionalElse:

 ELSE {$$=triple; emit("JUMP",0,0);} statement {backpatch1($2,-triple); $$=$2+1} |

 /\* empty \*/ { $$=triple; }

 ;

assign:

 IDENT sub ASSIGN exp

 { if ($1==0) {

 error("undefined");

 emit("ERROR",0,0);

 }

 else if ($2==0) {

 if (symtab[$1].size>0) {

 error("need subscript");

 emit("ERROR",0,0);

 }

 else emit(":=",$1,$4);

 }

 else if (symtab[$1].size==0) {

 error("cannot subscript");

 emit("ERROR",0,0);

 }

 else {

 emit("[]=",$1,$2);

 emit("\_",$4,0);

 }

 }

 ;

sub:

 LBRACK exp RBRACK

 { $$ = $2; } |

 /\* empty \*/

 { $$ = 0; }

 ;

exp:

 exp PLUS exp

 { $$ = -triple;

 emit("+",$1,$3);

 } |

 exp MINUS exp

 { $$ = -triple;

 emit("-",$1,$3);

 } |

 exp TIMES exp

 { $$ = -triple;

 emit("\*",$1,$3);

 } |

 exp DIVIDE exp

 { $$ = -triple;

 emit("/",$1,$3);

 } |

 LPAREN exp RPAREN

 { $$ = $2; } |

 NUMBER

 { $$ = -triple;

 emit("CON",$1,0);

 } |

 IDENT sub

 { $$ = -triple;

 if ($1==0) {

 error("undefined variable");

 emit("ERROR",0,0);

 }

 else if ($2==0) {

 if (symtab[$1].size>0) {

 error("need subscript");

 emit("ERROR",0,0);

 }

 else $$ = $1;

 }

 else if (symtab[$1].size==0) {

 error("cannot subscript");

 emit("ERROR",0,0);

 }

 else emit("=[]",$1,$2);

 }

 ;

%%

void basicinit( void) {

 code[0].opcode[0] = '\0';

 showStart = 1; /\* Index of next code slot that has yet to be dumped \*/

 triple = 1; /\* Index of first available code slot \*/

 symsize = 1; /\* Index of next available symbol table slot (0-th reserved) \*/

 errorNo = 0; /\* Number of errors detected \*/

 warnNo = 0; /\* Number of warnings given \*/

}

void enter(char \*name, int size) {

 char \*p;

 if (symsize<SYMMAX) { symtab[symsize].size = size;

 symtab[symsize].countRead = 0;

 symtab[symsize].countWrite = 0;

 p = symtab[symsize++].name;

 while (\*p++ = \*name++);

 }

 else fatal("symbol table too large");

}

void backpatch(int trip, int val) { /\* fix a conditional jump \*/

 code[trip].second = val;

}

void backpatch1(int trip, int val) { /\* backpatch the first field; unconditional jump \*/

 code[trip].first = val;

}

void emit(char \*opcode, int first, int second) { /\*generates code and handles var count\*/

 char \*p;

 if (triple<CODEMAX) {

 if ( strcmp(opcode, "con") ) { /\* not a constant, so need to do further checks \*/

 if (second>0) /\* 2nd var only used for read \*/

 symtab[second].countRead++; /\* rvalue; note: in a[a1]:=b; a1 is a read \*/

 if ( !strcmp(opcode, ":=") || !strcmp(opcode, "[]=") )

 symtab[first].countWrite++; /\* lvalue \*/

 else if (first>0)

 symtab[first].countRead++; /\* rvalue; first operand \*/

 }

 code[triple].first = first;

 code[triple].second = second;

 p = code[triple++].opcode;

 while (\*p++ = \*opcode++);

 }

 else fatal("code table is too large\n");

}

void ok(int extra) {

 while ((yychar!=WHILE) && (yychar!=SEMICOLON) && (yychar!=extra) &&

 (yychar>0)) yychar = yylex();

}

void fatal(char const \*msg) {

 printf("%s\n",msg);

 dumpcode();

}

void dumpcode(void) {

 int i;

 printf("\n");

 for (i=showStart;i<triple;i++)

 printf("%4d : %8s %4d %4d\n",i,code[i].opcode,code[i].first,code[i].second);

 showStart = triple;

}

void countReport(void) {

 int i;

 printf("\n\*\*\* NUMBER OF ERRORS FOUND: %d \*\*\* \n", errorNo);

 printf( "\*\*\* NUMBER OF WARNINGS: %d \*\*\* \n", warnNo);

 /\* dump out symbol table with reference counts \*/

 for (i=1; i<symsize; i++)

 if (symtab[i].size>0)

 printf("%s[%d], # of read ref = %d, # of write ref = %d\n",

 symtab[i].name, symtab[i].size, symtab[i].countRead,

 symtab[i].countWrite);

 else

 printf("%s, # of read ref = %d, # of write ref = %d\n",

 symtab[i].name, symtab[i].countRead, symtab[i].countWrite);

}

void error(char const \*s) {

 printf("\nError: %s\n", s);

 errorNo++;

}

void warning(char const \*s) {

 printf("\nWarning: %s\n", s);

 warnNo++;

}

void yyerror(char const \*s) {

 printf("%s\n", s);

}

#include "while.lex.c"

int main(int argc, char \*\*argv ) {

 int result;

 ++argv, --argc; /\* skip over program name \*/

 if ( argc > 0 ) yyin = fopen( argv[0], "r" );

 else yyin = stdin;

 result = yyparse();

 system("pause()");

 return(result);

}

%{

/\*////////////////////////////////////////////////////////////////////////

// LEX for a simple language //

// //

// FILENAME: while.l //

// //

// Author(s): Charles E. Hughes //

//

////////////////////////////////////////////////////////////////////////\*/

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <math.h>

#include <ctype.h>

#include <limits.h>

void setid(void);

void setnum(void);

%}

L [A-Za-z]

D [0-9]

ID {L}({L}|{D})\*

NUM {D}+

%%

">=" {ECHO; yylval = (int) '+'; return(RELOP);}

"<=" {ECHO; yylval = (int) '-'; return(RELOP);}

"="|"#"|">"|"<" {ECHO; yylval = (int) yytext[0]; return(RELOP);}

"+" {ECHO; return(PLUS);}

"-" {ECHO; return(MINUS);}

"\*" {ECHO; return(TIMES);}

"/" {ECHO; return(DIVIDE);}

"{" {ECHO; return(LBRACE);}

"}" {ECHO; return(RBRACE);}

"(" {ECHO; return(LPAREN);}

")" {ECHO; return(RPAREN);}

"[" {ECHO; return(LBRACK);}

"]" {ECHO; return(RBRACK);}

";" {ECHO; return(SEMICOLON);}

"," {ECHO; return(COMMA);}

":=" {ECHO; return(ASSIGN);}

"while" {ECHO; return(WHILE);}

"do" {ECHO; return(DO);}

"end" {ECHO; return(END);}

"if" {ECHO; return(IF);}

"then" {ECHO; return(THEN);}

"else" {ECHO; return(ELSE);}

{NUM} {ECHO; setnum(); return(NUMBER);}

{ID} {ECHO; setid(); return(IDENT);}

[ \t]\* {ECHO;}

[\n]+ {ECHO; dumpcode();}

. {ECHO; printf("\nunrecognizable character\n");

 return(BAD);}

%%

void setnum(void)

{ int i;

 yylval = 0;

 i = 0;

 while (yytext[i]) /\* convert string to int \*/

 yylval = yylval\*10 + ( (int) yytext[i++] - (int) '0' );

}

void setid(void)

{ char \*p,\*q,\*r;

 p = idname; /\* used to communicate string to syntax analyzer \*/

 q = yytext; /\* string found in input \*/

 r = symtab[0].name; /\* new symbol starts i zero-th slot \*/

 if (yyleng>=IDLENGTH)

 yytext[IDLENGTH-1] = '\0'; /\* null termination of string makes copy safe \*/

 while (\*r++ = ( \*p++ = \*q++ )); /\* copy new symbol into table \*/

 yylval = symsize; /\* search bottom up -- small table so linear okay \*/

 while ( strcmp(symtab[--yylval].name, idname) ); /\* always succeeds \*/

}

int yywrap(void)

{

 return(1);

}

===============================================

GOOD PROGRAM

===============================================

var1,var2,a[5],b[7];

var1 := var2+10; var1 := 9;

var1 := 1;

while var1<5 do

{ a[var1] := var1\*var1 + 1;

 b[var1] := a[var1] + var2;

 var1 := var1+1;

}

end

================================================

BAD PROGRAM

================================================

,,var1,:,var2,a[5],b[7];

var1 := var2+10++; var1 := 9;

var1 := 1;

junk while var1<<5 do

{ a[var1] := var1\*var1 + 1;

 b[var1] := a[var1] + var2;

 var1 := var1+1;

}

var1 : = { var1 := 4; }

end

var1,var2,a[5],b[7];

var1 := var2+10; var1 := 9;

 1 : CON 10 0

 2 : + 2 -1

 3 : := 1 -2

 4 : CON 9 0

 5 : := 1 -4

var1 := 1;

 6 : CON 1 0

 7 : := 1 -6

while var1<5 do

 8 : CON 5 0

 9 : - 1 -8

 10 : JNM -9 0

{ a[var1] := var1\*var1 + 1;

 11 : \* 1 1

 12 : CON 1 0

 13 : + -11 -12

 14 : []= 3 1

 15 : \_ -13 0

 b[var1] := a[var1] + var2;

 16 : =[] 3 1

 17 : + -16 2

 18 : []= 4 1

 19 : \_ -17 0

 var1 := var1+1;

 20 : CON 1 0

 21 : + 1 -20

 22 : := 1 -21

}

 23 : JUMP -8 0

end

 24 : EXIT 0 0

 1 : CON 10 0

 2 : + 2 -1

 3 : := 1 -2

 4 : CON 9 0

 5 : := 1 -4

 6 : CON 1 0

 7 : := 1 -6

 8 : CON 5 0

 9 : - 1 -8

 10 : JNM -9 -24

 11 : \* 1 1

 12 : con 1 0

 13 : + -11 -12

 14 : []= 3 1

 15 : \_ -13 0

 16 : =[] 3 1

 17 : + -16 2

 18 : []= 4 1

 19 : \_ -17 0

 20 : CON 1 0

 21 : + 1 -20

 22 : := 1 -21

 23 : JUMP -8 0

 24 : EXIT 0 0

\*\*\* NUMBER OF ERRORS FOUND: 0 \*\*\*

\*\*\* NUMBER OF WARNINGS: 0 \*\*\*

var1, # of read ref = 7, # of write ref = 4

var2, # of read ref = 2, # of write ref = 0

a[5], # of read ref = 1, # of write ref = 1

b[7], # of read ref = 0, # of write ref = 1